1. **Compare Props and State**
    - **Answer**:
        - **Definition**:
          Props (short for properties) are read-only, passed from parent to child components to configure them. State is managed within a component and can change over time.
        - **Example**:

```
// Props example
const Child = (props) => <h1>{props.message}</h1>;
const Parent = () => <Child message="Hello!" />;

// State example
class Counter extends React.Component {
  state = { count: 0 };
  increment = () => this.setState({ count: this.state.count + 1 });
  render() {
    return <button onClick={this.increment}>{this.state.count}</button>;
  }
}
```

2. **Develop React Component to Toggle Messages**
    - **Answer**:
        - **Definition**:
          Toggle between different messages on button click using component state.
        - **Example** (Code):

```
import React, { useState } from 'react';

const ToggleMessage = () => {
  const [message, setMessage] = useState("Message 1");
```

```
  const toggle = () => {
    setMessage(message === "Message 1" ? "Message 2" : "Message 1");
  };

  return (
    <div>
      <p>{message}</p>
      <button onClick={toggle}>Toggle Message</button>
    </div>
  );
};
```

3. **Pass a Method from Parent to Child using Props**
    - **Answer**:
        - **Definition**:
          Pass methods from the parent component as props, allowing the child component to execute the parent's methods.
        - **Example** (Code):

```
const Parent = () => {
  const parentMethod = () => alert("Hello from Parent!");

  return <Child callParent={parentMethod} />;
};

const Child = (props) => {
  return <button onClick={props.callParent}>Call Parent</button>;
};
```

4. **Process of Initialization of State in Class Component**
    - **Answer**:

- **Definition**:
  In class components, the state is initialized within the constructor or directly in the class body as an object.
- **Example** (Code):

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return <div>{this.state.count}</div>;
  }
}
```

5. **Apply Lifecycle Method to Lock a Message when State Changes**
   - **Answer**:
     - **Definition**:
       Use componentDidUpdate lifecycle method to lock a message after state updates.
     - **Example** (Code):

```
class MessageLocker extends React.Component {
  state = { message: "Initial message", locked: false };

  componentDidUpdate(prevProps, prevState) {
    if (prevState.message !== this.state.message && !this.state.locked) {
      this.setState({ locked: true });
    }
  }
```

```
  updateMessage = () => {
    if (!this.state.locked) {
      this.setState({ message: "Updated message" });
    }
  };

  render() {
    return (
      <div>
        <p>{this.state.message}</p>
        <button onClick={this.updateMessage}>Update Message</button>
      </div>
    );
  }
}
```

6. **Compare Default Props & PropTypes**
    - **Answer**:
        - **Definition**:
          DefaultProps define default values for props if they are not provided, whereas PropTypes check the type of the props passed to a component.
        - **Example** (Code):

```
import PropTypes from 'prop-types';

const Greeting = ({ name }) => <h1>Hello, {name}!</h1>;

Greeting.defaultProps = { name: "Guest" };
Greeting.propTypes = { name: PropTypes.string };

// If no "name" is passed, it defaults to "Guest"
```

7. **Compare State Initialization in Class and Functional Component**
    - **Answer**:
        - **Definition**:
          In class components, state is initialized using this.state, while in functional components, state is initialized using useState hook.
        - **Example** (Code):

```
// Class Component
class MyComponent extends React.Component {
  state = { count: 0 };
}

// Functional Component
const MyComponent = () => {
  const [count, setCount] = useState(0);
};
```

8. **Compare Different Types of Events Handled in React**
    - **Answer**:
        - **Definition**:
          React handles events like onClick, onChange, onSubmit, onMouseEnter, and more. They behave similarly to native DOM events but work cross-browser.
        - **Example** (Code):

```
const Button = () => <button onClick={() => alert("Clicked!")}>Click Me</button>;

const Input = () => <input onChange={(e) => console.log(e.target.value)} />;
```

9. **Benefits of Using Arrow Function in Event Handlers**
    - **Answer**:
        - **Definition**:
          Arrow functions automatically bind this to the enclosing context, avoiding the need for explicit bind in class components.

- **Example** (Code):

```
class MyComponent extends React.Component {
  handleClick = () => {
    console.log(this);
  };

  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

10. **Sample Program to Get Username and Password Using Controlled Components**

- **Answer**:
  - **Definition**:
    Controlled components store input values in the component's state and update state on every input change.
  - **Example** (Code):

```
const LoginForm = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Username:", username, "Password:", password);
  };

  return (
    <form onSubmit={handleSubmit}>
```

```
        <input value={username} onChange={(e) => setUsername(e.target.value)}
placeholder="Username" />
        <input value={password} onChange={(e) => setPassword(e.target.value)}
placeholder="Password" type="password" />
        <button type="submit">Login</button>
      </form>
    );
};
```

---

11. Design a Counter Application that Increments & Decrements Count

- **Answer**:
    - **Definition**:
      Design a counter that uses state to track the count value. The count should not go below 0. Describe how the message state updates within this component.
    - **Example** (Code):

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);
  const [message, setMessage] = useState("");

  const increment = () => {
    setCount(count + 1);
    setMessage("Count incremented!");
  };

  const decrement = () => {
    if (count > 0) {
      setCount(count - 1);
      setMessage("Count decremented!");
    } else {
```

```
      setMessage("Count cannot go below 0.");
    }
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
      <p>{message}</p>
    </div>
  );
};
```

---

12. Implement a Parent Component that Passes a List to Child and Allows Item Removal

- **Answer**:
  - **Definition**:
    The parent maintains a list and passes it as props to the child. The child can remove an item, which updates the parent's state.
  - **Example** (Code):

```
// Parent Component
const Parent = () => {
  const [items, setItems] = useState(["Item 1", "Item 2", "Item 3"]);

  const removeItem = (index) => {
    const newItems = items.filter((_, i) => i !== index);
    setItems(newItems);
  };

  return <Child items={items} removeItem={removeItem} />;
```

```
};

// Child Component
const Child = ({ items, removeItem }) => {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>
          {item} <button onClick={() => removeItem(index)}>Remove</button>
        </li>
      ))}
    </ul>
  );
};
```

- **Changes in Parent**:
  When an item is removed in the child, the parent's state is updated, reflecting the new list in the UI.

---

13. Illustrate Component Lifecycle with Examples

- **Answer**:
  - **Definition**:
    React component lifecycle methods include mounting, updating, and unmounting phases like componentDidMount, componentDidUpdate, and componentWillUnmount.
  - **Example** (Code):

```
class LifeCycleDemo extends React.Component {
  componentDidMount() {
    console.log("Component mounted!");
  }

  componentDidUpdate(prevProps, prevState) {
```

```
    console.log("Component updated!");
  }

  componentWillUnmount() {
    console.log("Component will unmount!");
  }

  render() {
    return <div>LifeCycle Demo</div>;
  }
}
```

---

14. Develop a Bank Application (Controlled & Uncontrolled Components)

- **Answer**:
  - **Definition**:
    Controlled components manage their state using React state, whereas uncontrolled components rely on direct DOM manipulation (e.g., refs).
  - **Example** (Code):

```
import React, { useState, useRef } from 'react';

const BankApp = () => {
  const [balance, setBalance] = useState(1000);
  const amountRef = useRef();

  const deposit = () => {
    const amount = parseFloat(amountRef.current.value);
    if (!isNaN(amount)) setBalance(balance + amount);
  };

  const withdraw = () => {
```

```
    const amount = parseFloat(amountRef.current.value);
    if (!isNaN(amount) && amount <= balance) setBalance(balance - amount);
  };

  return (
    <div>
      <h1>Bank Application</h1>
      <p>Balance: ${balance}</p>
      <input ref={amountRef} type="number" placeholder="Enter amount" />
      <button onClick={deposit}>Deposit</button>
      <button onClick={withdraw}>Withdraw</button>
    </div>
  );
};
```

---

15. Illustrate Controlled & Uncontrolled Components in a Form

- **Answer**:
  - **Definition**:
    Controlled components manage form inputs through React state, while uncontrolled components access values directly via the DOM (e.g., using refs).
  - **Example** (Code):

```
import React, { useState, useRef } from 'react';

const ContactForm = () => {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const messageRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
```

```
    console.log("Name:", name);
    console.log("Email:", email);
    console.log("Message:", messageRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* Controlled Components */}
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Name"
      />
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Email"
      />

      {/* Uncontrolled Component */}
      <textarea ref={messageRef} placeholder="Message"></textarea>

      <button type="submit">Submit</button>
    </form>
  );
};
```

- **Explanation**:
  Controlled components (name, email) ensure real-time validation and tracking, while the uncontrolled component (message) provides a simpler setup, useful when form control is unnecessary.